

## Programmation Orientée Objet - Licence TIS CM3/9

Lancelot PECQUET

Lancelot.Pecquet@math.univ-poitiers.fr



Poitiers, le 23/01/2006

- 1 Introduction à l'héritage
- 2 Application en Java
- 3 Polymorphisme d'héritage et redéfinition
- 4 Encapsulation et héritage
- 5 Classe Object

## Rappel sur la séance précédente

La fois précédente, nous avons vu :

- 1 structure d'une classe
- 2 constructeurs
- 3 encapsulation
- 4 conversion en chaîne et égalité
- 5 gestion mémoire, destructeurs et composition

Aujourd'hui, nous voyons en détail la notion d'héritage.

## Définition de l'héritage

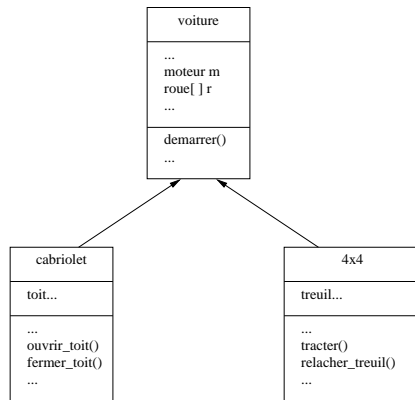
### Définition

L'héritage est un mécanisme permettant de dériver une nouvelle classe (dite fille) d'une classe (dite mère) en ne définissant que ce qui est ajouté ou modifié (redéfinition de méthodes).

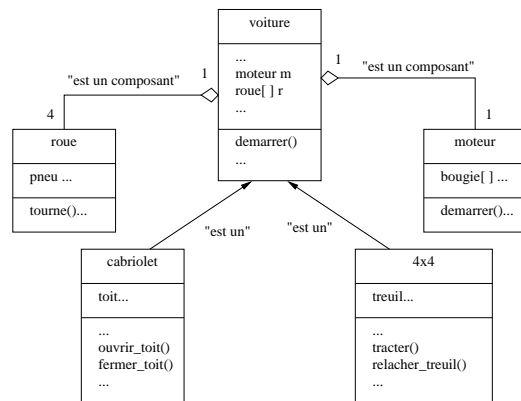
### Avantages

- code réutilisable facilement
- seul ce qui est spécifique doit être (re)défini
- l'encapsulation est préservée

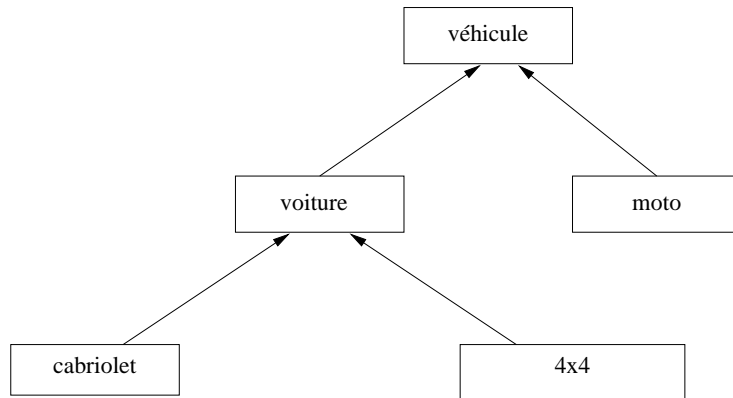
## Exemple : voiture, cabriolet, 4 × 4 . .



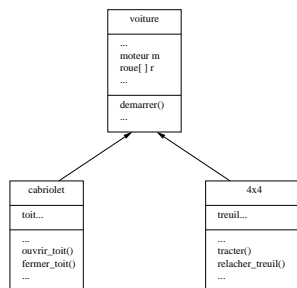
## Distinction entre héritage et agrégation/composition



## Arbre d'héritages successifs



## Notion d'héritage multiple



Un 4 × 4 décapotable ? Héritage multiple (ou agrégation ?)...

## Mot-clé extends : exemple d'un pixel

### Fichier point.java

```
1 class point{
2   private int x,y;           // champs
3   ...
4   public final int get_x(){return x;} // accesseurs et mutateurs
5   ...
6   public point(int x, int y){set_x(x); set_y(y);} // constructeurs
7 }
```

### Fichier pixel.java

```
1 class pixel extends point{ // mot-clé extends
2   private int R,G,B;       // champs nouveaux
3   ...
4   public final int getR(){return R;} // nouvelles methodes
5   ...
6 }
```

## Mot-clé super et constructeurs

Le constructeur de pixel utilise celui de point :

```
1 class pixel extends point{
2   ...
3   public pixel(int x, int y, int R, int G, int B){
4     super(x,y); // invocation du constructeur de point
5     setR(R);
6     setG(G);
7     setB(B);
8   }
9   ...
10 }
```

## Questions ?



## Polymorphisme de l'héritage

### Définition

Le **polymorphisme** permet d'appliquer une même méthode de la classe mère à ses héritières de manière transparente.

### Exemple (Main.java)

```
1 pixel p = new pixel(3,2,255,255,255); // un pixel blanc au point (3,2)
2 System.out.println(p.get_x()); // invocation de get_x() depuis la classe fille
```

## Qui est qui ?

### Règle

Une instance doit toujours être totalement spécifiée → une lvalue (à gauche du signe =) doivent être parente (au sens large) de la rvalue correspondante (à droite du signe =).

### Exemple

```
1 pixel P1 = new pixel(3,2,255,255,255); // usage classique
2 point q1 = new point(3,2); // usage classique
3 point q2 = P1; // convient car le point est totalement specifie
4 pixel P2 = q1; // ERREUR car RGB non specifie
```

## Redéfinition d'une méthode

### Définition dans la classe mère (point.java)

```
1 class point{...
2   public String toString(){return "(" + get_x() + "," + get_y() + " "};
3   ..}
```

### Redéfinition dans la classe fille (pixel.java)

```
1 class pixel extends point{...
2   public String toString(){
3     return super.toString() + "couleur=("
4       + getR() + "," + getG() + "," + getB() + ")";
5   }
6   ...}
```

## Redéfinition d'une méthode (suite)

### Exemple d'utilisation (Main.java)

```
1 ...  
2 point p = new point(3,2);  
3 pixel q = new pixel(3,2,128,25,13);  
4 System.out.println(p);  
5 System.out.println(q);  
6 ...
```

### Résultat (console)

```
1 (3,2)  
2 (3,2) couleur=(128,25,13)
```

## Questions ?



## Champs protected

### point.java

```
1 class point{...
2   protected int x,y;
3 }
```

### pixel.java

```
1 class pixel extends point{...
2   public void reset(){x=0; y=0;} // acces direct au champ par une heritiere
3 }
```

Parfois plus efficace mais perte d'encapsulation !

## Champs protected (suite)

### Main.java

```
1 ...
2 pixel p = new pixel(3,2,255,255,0);
3 p.reset(); // fonctionne
4 p.x=0; // erreur car Main n'herite pas de point (passe si package...)
5 ...
```

## Méthodes protected

### point.java

```
1 class point{...
2   private int x,y;
3   protected void reset(){x=0;y=0;}
4   ...}
```

### pixel.java

```
1 class pixel extends point{...
2   reset(); // fonctionne
3   p.x=0; // erreur car x est prive dans point (passe si package...)
4   ...}
```

Pas de perte d'encapsulation

## Classes final

### Déclaration final de la classe point

```
1 final class point{...}
```

### La classe pixel final ne peut être héritée

```
1 class pixel extends point{...} // erreur
```

## Méthodes final

### Méthode final de la classe point

```
1 class point {...  
2   public final int get_x(){return x;}  
3   ...}
```

### La méthode final ne peut être redéfinie

```
1 class pixel extends point {...  
2   public int get_x(){return 0;} // erreur  
3   ...}
```

## Pourquoi des méthodes final dans les constructeurs ?

### Supposons

- le constructeur  $C()$  de la classe  $C$  utilise une méthode  $f$
- la classe  $C'$  dérive de  $C$
- la méthode  $f$  est redéfinie en  $f'$  dans  $C'$

### Alors

- le constructeur de  $C'$  appelle `super()` (i.e.  $C()$ )
- c'est  $f'$  qui est invoqué par  $C()$  **alors que l'instance de la superclasse n'est même pas encore construite** ( $\neq C++$ )

## Questions ?



## Classe `Object`

- toutes les classes dérivent d'`Object`
- cet héritage est transparent (pas de `extends Object...`)
- `Object` n'a pas d'ascendant

## Méthodes de la classe `Object`

- `public boolean equals(Object x)`
- `public int hashCode()`
- `protected Object clone() throws CloneNotSupportedException`
- `public final Class getClass()`
- `public void finalize() throws Throwable`
- `public String toString()`

## `public boolean equals(Object x)`

- égalité par référence (par défaut, opérateur `==`)
- destinée à être redéfinie → sémantique adéquate
- ex : `String.equals()` : égalité caractère par caractère

## `public int hashCode()`

- définit une fonction de hachage pour les tables
- destinée à être redéfinie → sémantique adéquate
- ex : `String.hashCode()` : renvoie la même valeur pour deux chaînes égales, caractère par caractère :

$$s[0] \times 31^{n-1} + s[1] \times 31^{n-2} + \dots + s[n-1]$$

## `protected Object clone() throws CloneNotSupportedException`

- permet de créer des copies de l'objet courant (on y reviendra)
- déclenche une `CloneNotSupportedException` en cas de problème

## `public final Class getClass()`

- renvoie la classe exacte de l'objet courant (celle du constructeur ayant servi à créer l'objet, même si celui a été casté)
- méthode `final` : ne peut pas être redéfinie

## `public void finalize() throws Throwable`

- destructeur par défaut
- peut être redéfini pour nettoyer fichiers, connexions. . .
- déclenche un `Throwable` (interruption) en cas de problème

## public String toString()

- renvoie une chaîne de caractères représentant l'objet courant
- par défaut : nom de classe + @ + hashCode()
- destinée à être redéfinie → sémantique adéquate
- ex : `point.toString()`
- appelée automatiquement par `System.out.println()`

## Quelques branches issues d'Object

```
Object
  Boolean
  Character
  Number
    Float
    Integer
    ...
  String
  Thread
  Throwable
  Error
    AssertionError
    LinkageError
    ...
  Exception
    RuntimeException
      ArithmeticException
      NullPointerException
      ...
```

## Questions ?



## Conclusion

Aujourd'hui, nous avons vu :

- ① Introduction à l'héritage
- ② Application en Java
- ③ Polymorphisme d'héritage et redéfinition
- ④ Encapsulation et héritage
- ⑤ Classe `Object`

La séance prochaine, nous verrons :

- classes abstraites
- interfaces et héritage multiple